📓 Taller de Arquitectura de Software

Refactorización de Monolito – Encuestas Espagueti

- Juan Sebastian Osorio Fierro
- Daniel Steve Fontalvo Matiz
- Leonado Fabio Perez Bermudez
- Juan Camilo Cruz Pardo

🔘 FASE 2 – Análisis de Anti-Patrones y Malas Prácticas

🎯 Objetivo

Analizar el sistema monolítico e identificar anti-patrones y malas prácticas clasificadas por:

- Seguridad
- Arquitectura
- SOLID
- Clean Code
- Manejo de errores
- Tipado
- Rendimiento

🔎 ACTIVIDAD 2.1 – Análisis del Backend (Spring Boot)

Archivo analizado:

EncuestaController.java

🔐 1. Seguridad

❌ 1.1 SQL Injection (Concatenación de SQL)

📍 Ubicación

- Método `crear()`
- Método `encuesta()`
- Método `votar()`

📌 Código Problemático

```
String sql = "INSERT INTO encuestas(pregunta, si_count, no_count) VALUES ('" +
pregunta + "', 0, 0)";
String sql = "SELECT id, pregunta, si_count, no_count FROM encuestas WHERE id = "
+ id;
```

🔎 Problema

Las consultas SQL se construyen mediante concatenación de strings.

Esto permite:

- Manipulación del query
- Alteración de la lógica SQL
- Inyección de código malicioso

⚠ Impacto

Vulnerabilidad crítica que compromete la seguridad del sistema.

❌ 1.2 Credenciales Hardcodeadas

```
private String URL = "jdbc:postgresql://db:5432/encuestas";
private String USER = "espagueti";
private String PASS = "espagueti";
```

- 🔎 Problema

1. Credenciales escritas directamente en el código fuente.
2. No se utilizan variables de entorno.

No se emplea application.properties o application.yml.

⚠ Impacto

Riesgo de exposición de credenciales.

Mala práctica DevOps.

No es portable a producción.

No cumple principios de configuración externa.

🏗 2. ARQUITECTURA ❌ 2.1 Ausencia de Arquitectura en Capas

El EncuestaController realiza:

Conexión a base de datos

Lógica de negocio

Validaciones

Construcción de SQL

Manejo de errores

🔎 Problema

No existen capas separadas como:

Controller

Service

Repository

DTO

Configuración de DataSource

Todo se encuentra en una sola clase.

⚠ Impacto

Alta acoplación

Difícil mantenimiento

Baja escalabilidad

Difícil testeo unitario

Violación del principio de separación de responsabilidades

🧱 3. PRINCIPIOS SOLID ❌ 3.1 Violación del SRP (Single Responsibility Principle)

El controlador:

Maneja lógica de negocio

Gestiona conexión a BD

Ejecuta SQL

Valida datos

Maneja errores

Construye respuestas

🔎 Problema

Un controlador debería:

Recibir la request

Delegar la lógica al Service

Retornar la respuesta

Actualmente hace múltiples responsabilidades.

⚠ Impacto

Código difícil de mantener

Baja cohesión

Alto acoplamiento

Difícil evolución futura

🧼 4. CLEAN CODE

❌ 4.1 Código Duplicado

Ejemplo de validaciones repetidas:

```
if (body == null || body.get("pregunta") == null ||
body.get("pregunta").toString().trim().length() < 3)
```

Este patrón se repite en múltiples métodos.

🎾 Problema

Validaciones repetidas.

No existe método reutilizable.

No se centraliza la lógica.

⚠ Impacto

Mayor probabilidad de errores.

Código más largo de lo necesario.

Difícil mantenimiento.

🚨 5. MANEJO DE ERRORES ❌ 5.1 Uso de printStackTrace() y return null

```
catch (Exception e) {
    e.printStackTrace();
}
return null;
```

🔎 Problema

Solo imprime el error en consola.

Retorna null.

No se devuelve un código HTTP adecuado.

No se utiliza @ExceptionHandler.

⚠ Impacto

Respuestas inconsistentes.

Mala práctica REST.

Dificulta monitoreo y trazabilidad.

Riesgo de NullPointerException en cliente.

🧾 6. TIPADO DEFICIENTE ❌ 6.1 Uso de Map sin Genéricos ni DTOs

```
public Map crear(@RequestBody Map body)
```

🔎 Problema

Uso de Map sin tipado genérico.

No existen DTOs.

No hay validación estructurada.

No hay clases de dominio.

⚠ Impacto

Pérdida de tipado fuerte.

Mayor riesgo de errores en tiempo de ejecución.

Código menos mantenible.

🚀 7. RENDIMIENTO ❌ 7.1 Conexión BD Recreada en Cada Llamada

```
private JdbcTemplate jdbc() {
    DriverManagerDataSource ds = new DriverManagerDataSource();
```

Cada vez que se ejecuta un endpoint:

Se crea un nuevo DataSource.

Se crea un nuevo JdbcTemplate.

Se genera una nueva conexión.

🔎 Problema

No se utiliza:

Pool de conexiones

Configuración centralizada de DataSource

⚠ Impacto

Ineficiencia

Bajo rendimiento bajo carga

No escalable

💻 ACTIVIDAD 2.2 – Análisis del Frontend (Angular)

Archivos analizados:

crear.component.ts

encuesta.component.ts

respuestas.component.ts

❌ 1. URL Hardcodeada this.http.post("http://localhost:8081/crear", ...)

🔎 Problema

No se usa environment.ts.

No hay configuración por entorno.

⚠ Impacto

No portable a producción.

Mala práctica de configuración.

❌ 2. Uso Directo de HttpClient en el Component constructor(private http: HttpClient)

🔎 Problema

No existe una capa Service intermedia.

Se viola el patrón de arquitectura Angular recomendado.

⚠ Impacto

Violación del patrón Service.

Lógica mezclada con presentación.

Dificulta pruebas unitarias.

❌ 3. Manipulación Directa del DOM document.getElementById(...)

🔎 Problema

Angular debe utilizar:

Data Binding

Directivas

Templates reactivos

Manipular el DOM directamente es un anti-patrón.

⚠ Impacto

Código menos mantenible.

Rompe el enfoque declarativo de Angular.

❌ 4. Uso Excesivo de any (r: any)

🔎 Problema

No hay interfaces tipadas.

No hay modelos de datos.

Se pierde el beneficio de TypeScript.

⚠ Impacto

Riesgo de errores.

Pérdida de autocompletado fuerte.

Código menos robusto.

❌ 5. Polling Manual con setInterval

Uso de:

setInterval(...)

En lugar de:

Observables

RxJS

AsyncPipe

⚠ Impacto

Código menos reactivo.

No aprovecha el paradigma Angular.

Posible consumo innecesario de recursos.

📌 CONCLUSIÓN TÉCNICA – FASE 2

El sistema presenta múltiples anti-patrones y malas prácticas:

Vulnerabilidad a SQL Injection

Credenciales hardcodeadas

Violación del principio SRP

Ausencia de arquitectura en capas

Manejo deficiente de errores

Bajo rendimiento por mala gestión de conexiones

Uso incorrecto de patrones en Angular

Se trata de un monolito funcional pero mal estructurado, lo que justifica una refactorización arquitectónica hacia:

Arquitectura en capas (Controller → Service → Repository)

Uso de DTOs

Configuración externa

Uso de Prepared Statements

Implementación de pool de conexiones

Separación adecuada en frontend (Service layer)